More OOP, (less oof)
ADTs

CSCI(41);

How are the terminal-based editors treating you?

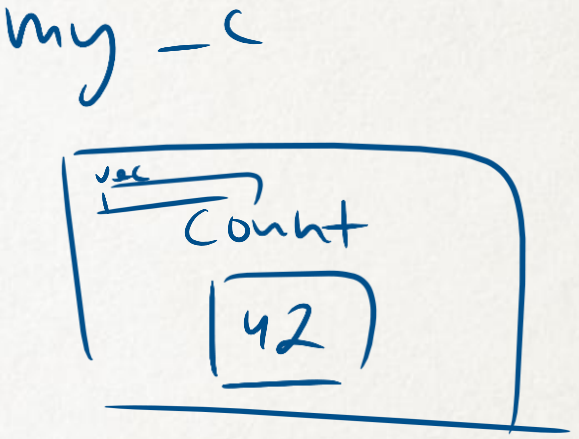
Recap: Here is a list of things that happen when an object is born, in the order that they occur:

class Counter {

private:

};

```
Counter my_c(42);
```



1. Memory gets allocated for the object
2. The appropriate constructor is called on that object to initialize it:
    1. The initializer list initializes all member variables mentioned
    2. Any **non-primitive** member variables not mentioned in the initializer list have their default constructor called
    3. The body of the constructor is executed to perform any additional init stuff

Primitive types: *int, bool, double, char, array. Anything that isn't a class.*

What I want you to be able to do

- Differentiate data structures from abstract data types
- Explain the implementation of the vector abstract data type
- Build Makefiles to compile programs
- Compare the relative amount of time and space that two programs use

But first: <sstream>

Let's look at the docs!

Let's build up a string

Let's extract stuff

Arrays & Dynamic Allocation

Review of arrays

- Array: “a variable with one name that holds a list of data items of the same type, each being directly accessible”
  - Each item is called an **element**
  - Directly accessible means that we can *index into* our array using the [] operator (like the .at() operator but *more dangerous*)
- E.g., int arr[50];

arr[2] = 42; arr[50]

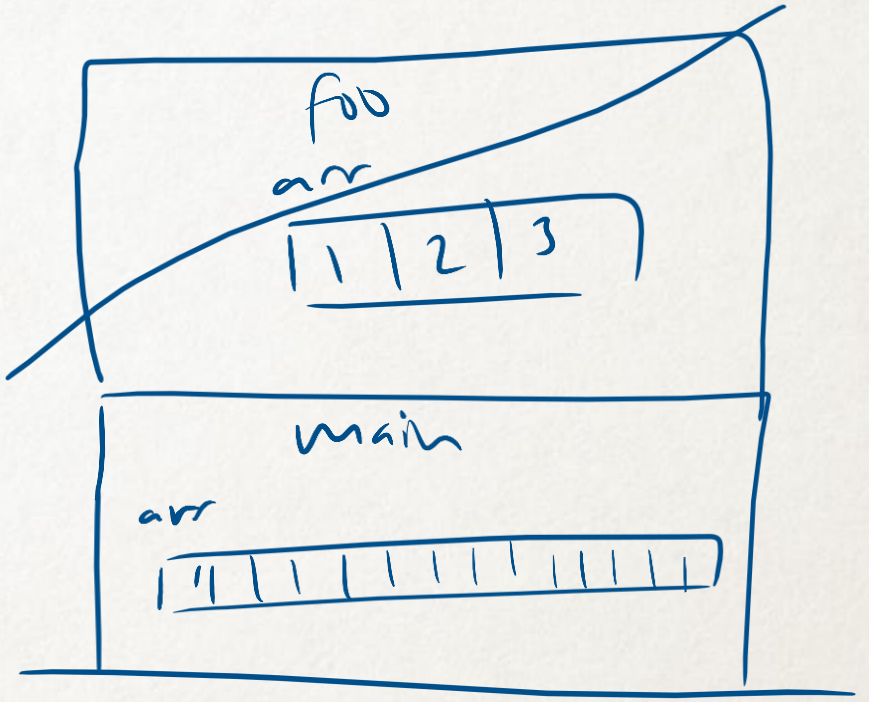
How to declare/initialize arrays on the stack

- That is, as a local variable

```
int main() {
    int arr[50];
    ... arr[0] = 1;
    foo();
    ...
}
```

```
void foo() {
    ...
    int arr[] =
        {1,2,3};
    ...
}
```

Stack diagram



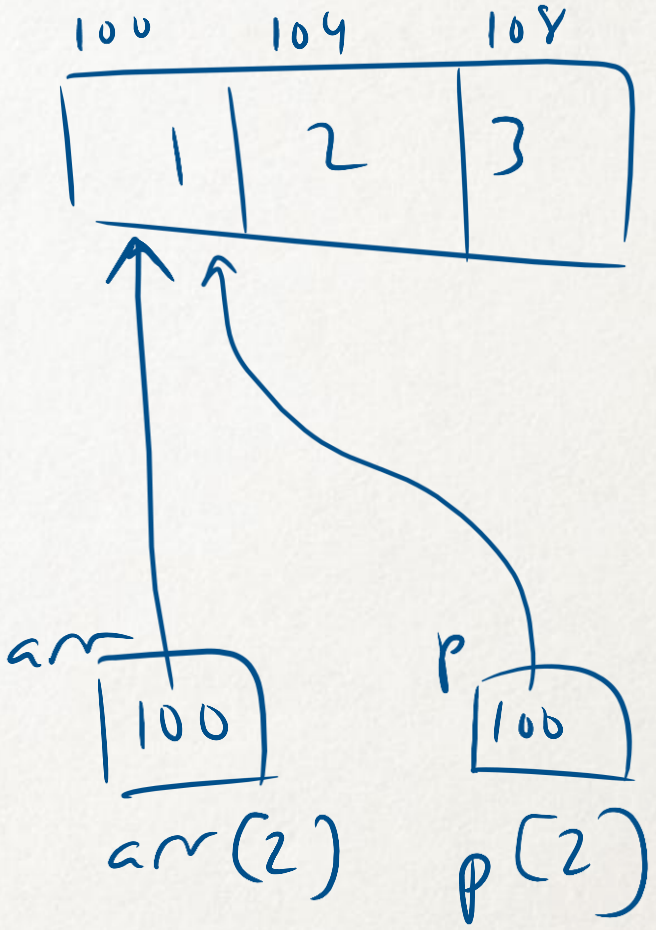
Arrays are pointers to their first elements

```
int arr[] = {1, 2, 3};
int* p = arr; // this totally works

// which means this:
int sum(int arr[], int size);

// can just as equally be this:
int sum(int* arr, int size);
```

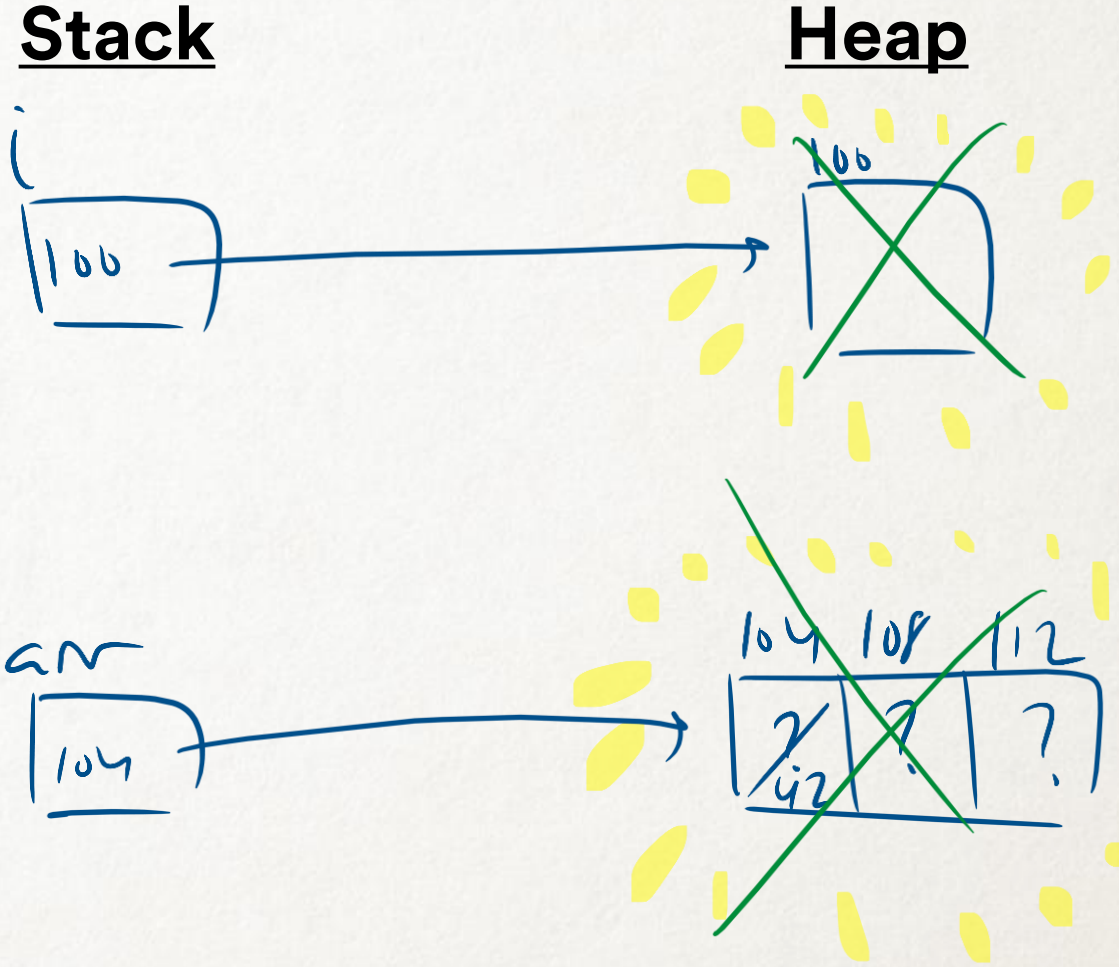
arr[1]
|||
*(arr + 1)



Creating and deleting single values & arrays on the heap

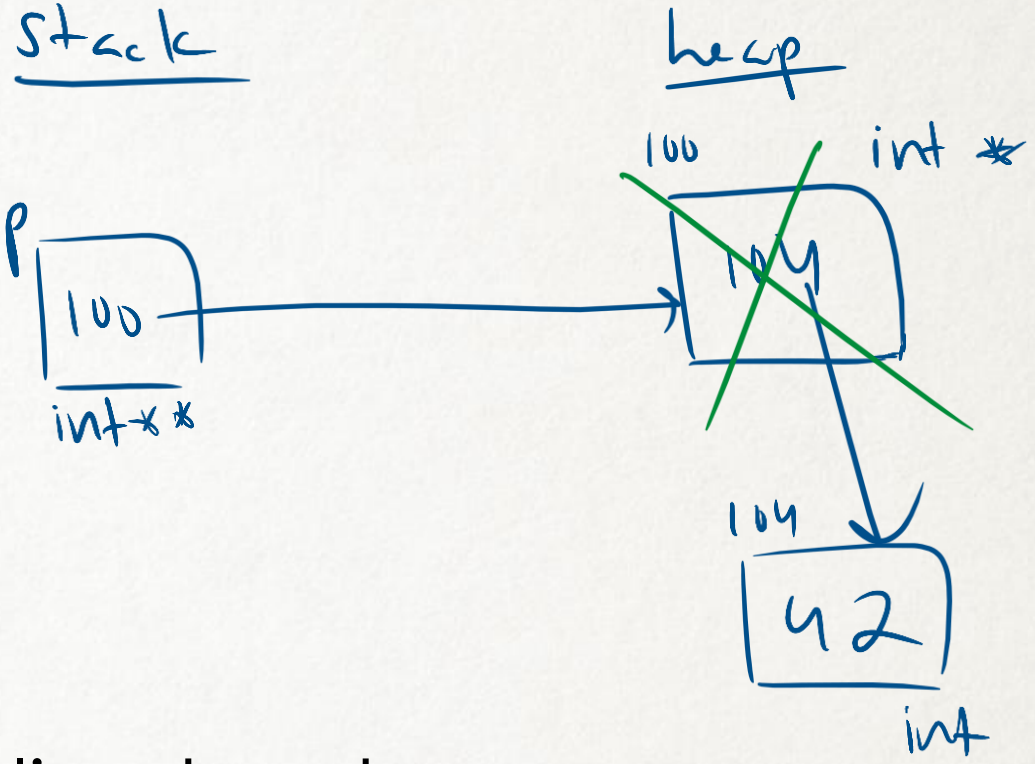
```
int* i = new int;
delete i;

int* arr = new int[3];
delete[] arr;
```




```
arr[0] = 42;
```

Crushed ICE



- Consider the following code:

```
int** p = new int*;
*p = new int;
**p = 42;
```

1. Draw the state of memory after the lines have been executed—make sure to specify which boxes are on the heap/stack
2. Draw what happens when the line **delete p;** is executed

delete *p;

Vectors and strings secretly use dynamically allocated arrays

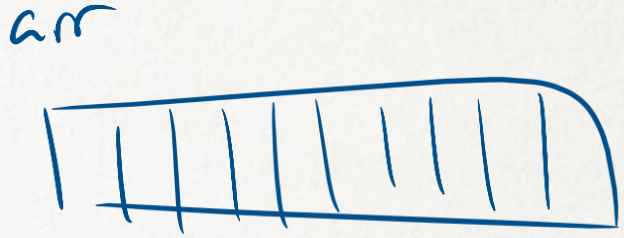
Algorithms & Data Structures

Algorithms

Space
time
> complexity

- An **algorithm** describes a sequence of steps to solve a computational problem or perform a calculation
  - A recipe!
- **Computational complexity**: the amount of resources used by a program
  - We only have so much **memory (space)** to do work with, and we only have so much **time** to wait for our programs to finish
- **Algorithm efficiency** is measured by computational complexity

Data Structures and ADTs

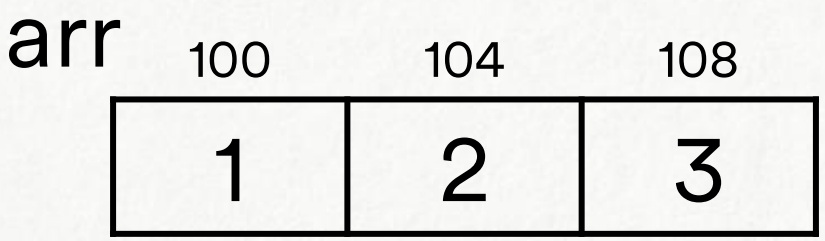


- A **data structure** is a way of organizing, storing, and performing operations on data
  - You know exactly what the memory looks like
  - Think array
- **Abstract data type (ADT)**: a data type described by predefined user *operations*
  - You don't know how it's implemented; you just know what you can do with it
  - Think vector
- Common operations on data structures and ADTs: access, update, search, insert, remove

↓ *All about the memory*

Data Structures vs. ADTs

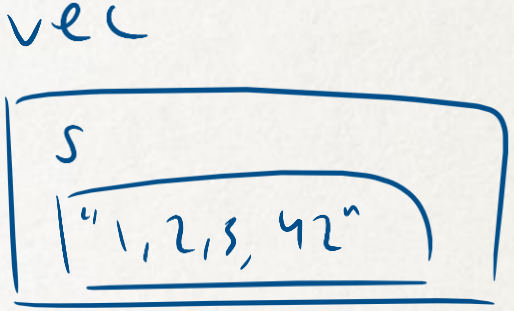
You know exactly what's going on in an array

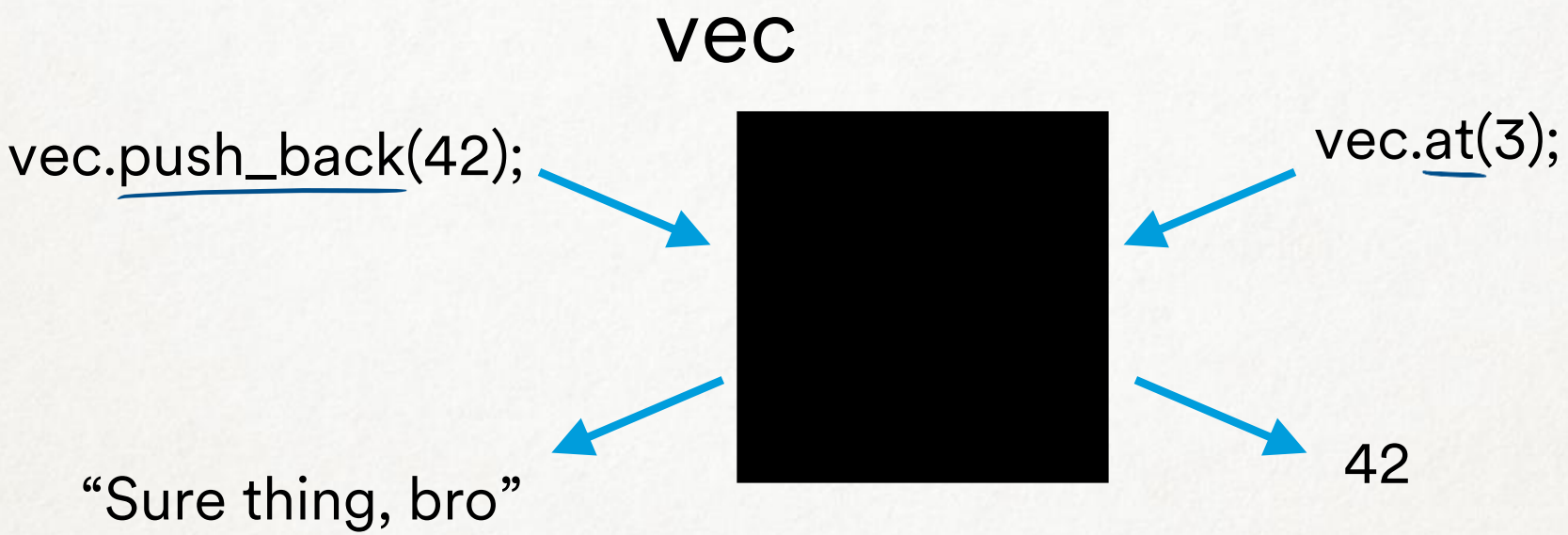


↓ *All about the operations*

Data Structures vs. ADTs

Whereas for vectors you've been kept in the dark.
For all you know, the internal representation is a string!





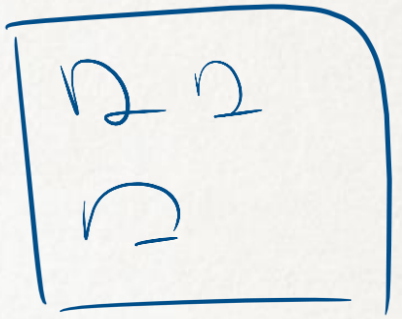
Data structures vs. ADTs

- What *makes* a vector a vector isn't that it has an array inside of it
  - You could use a funky-looking string, another data structure called a linked list (more on that later), etc.
  - The data structure that *implements* an ADT is not important
- What makes a vector a vector is that you can do .push_back(), .at(), .size(), etc., and it's guaranteed to do what you expect
- The underlying data/implementation for an ADT doesn't matter

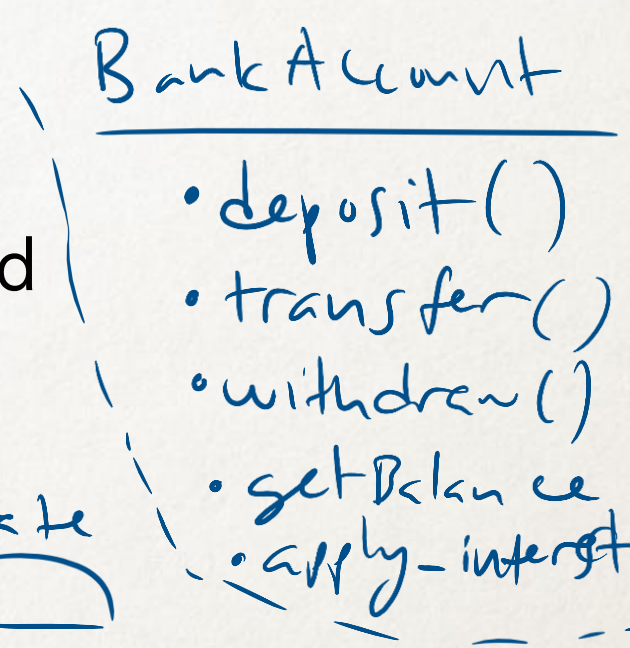
An abstract data type is an abstract *idea*, while a data structure is something concrete and tangible that can be used to *implement* an ADT

*ADTs **are implemented using** data structures.*

Crushed ICE



- Come up with a definition for a BankAccount ADT to represent your bank account *(just make a list, don't implement anything)*
  - What operations should be implemented for a BankAccount?
  - What data structures might you put inside a BankAccount to use while implementing the operations?



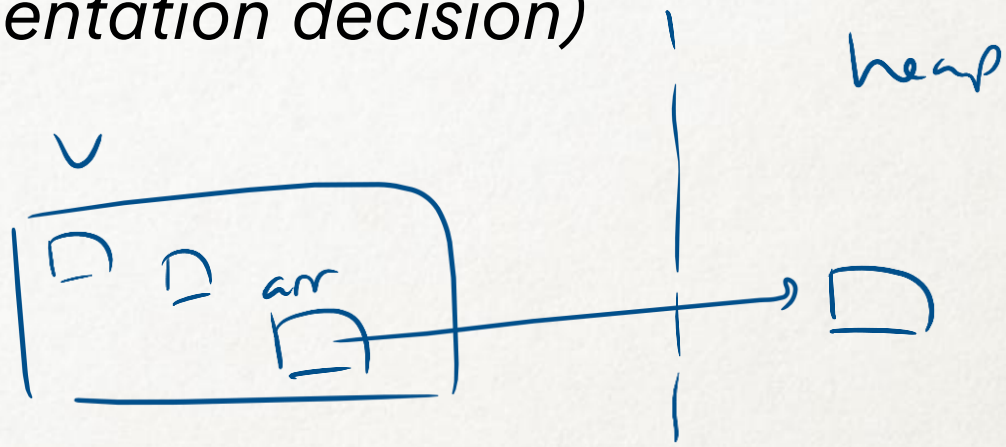
- Get as fancy as you want within the time period
  - E.g.: credit cards, loans, etc.

~~History~~ Memory is ~~written~~ allocated by the ~~victors~~ vectors

Our Vector ADT (your lab)

- push_back(val): add one element to the end of the vector, increasing the size by 1
- size(): get the current size of the Vector
- at(index): get the element held at a given index
- Start out empty, with room to hold one value
  - *(But that's just an implementation decision)*

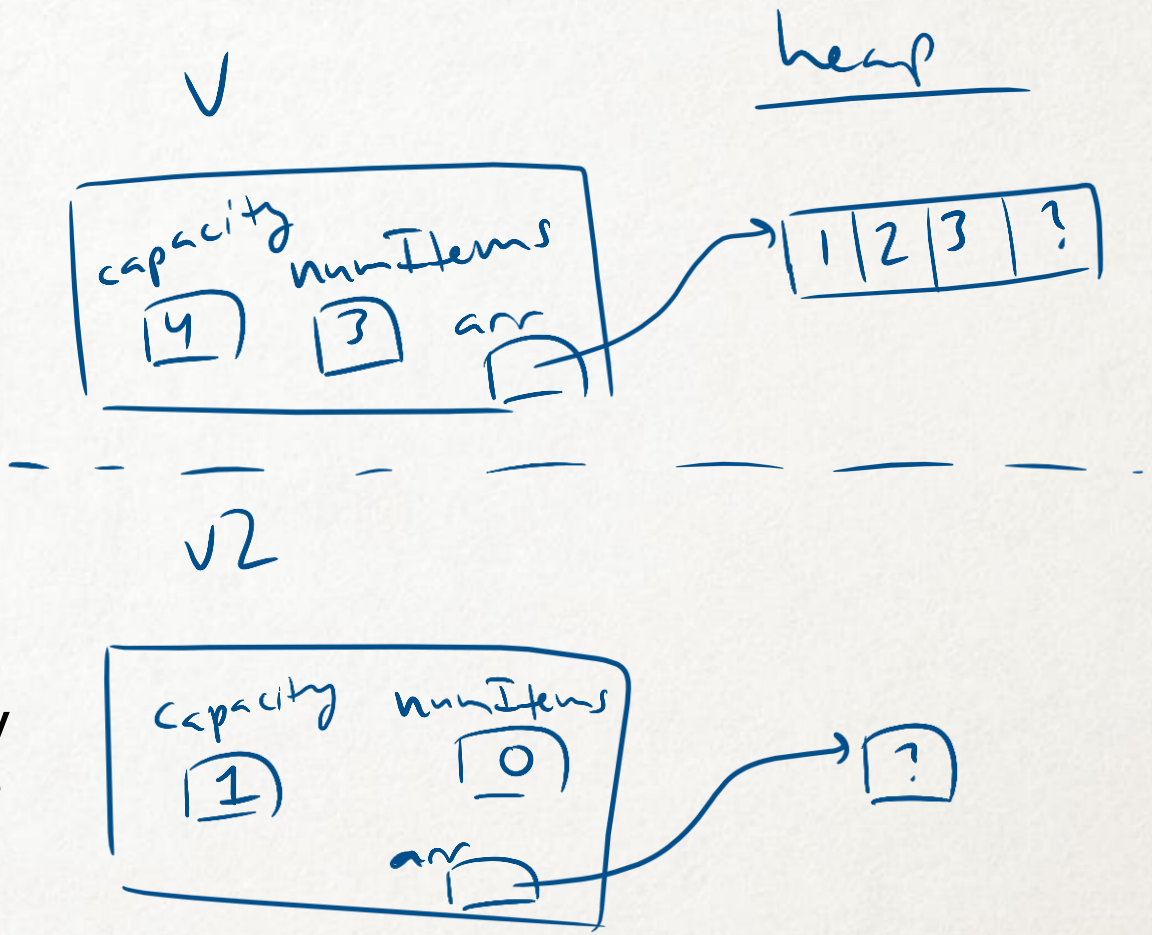
ar = new int[1];



Implementing vectors using arrays

Key idea for vectors: if our current array doesn't have enough room, we make a new one. This is why we need to use the heap!

- This is your lab, so I won't write code here!
- We'll save the elements in an array stored on the heap
- Private member variables: dynamic array (arr), current size of the vector (numItems), current array capacity (capacity)
- Constructor:
  - Initialize `numItems` to 0, `capacity` to 1, and make an array `arr` on the heap with size `capacity`
  - Let's draw this



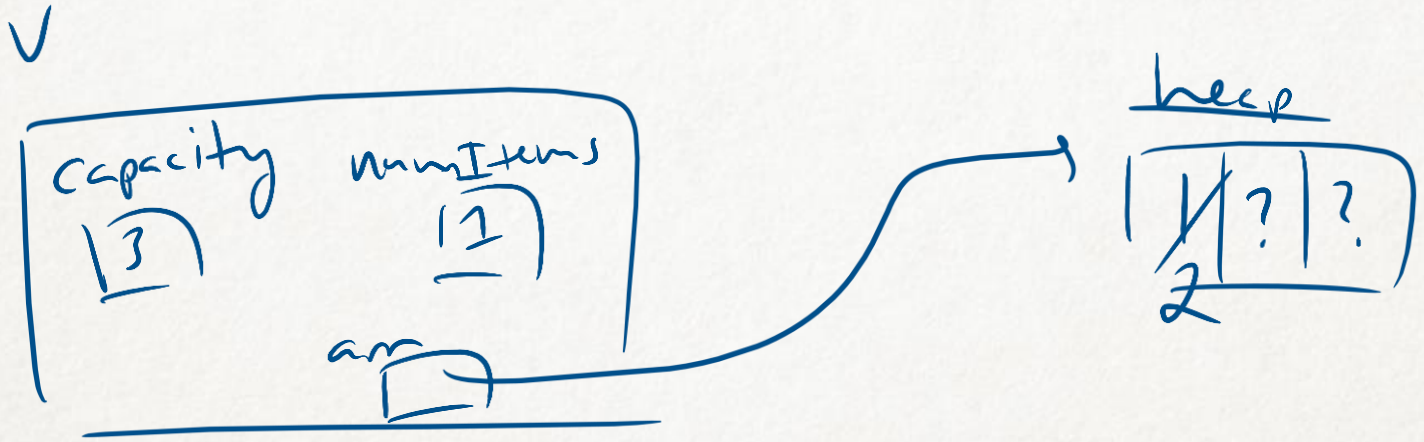
.at(...)

v.at(0) ⟶ 1

v.at(0) = 2;

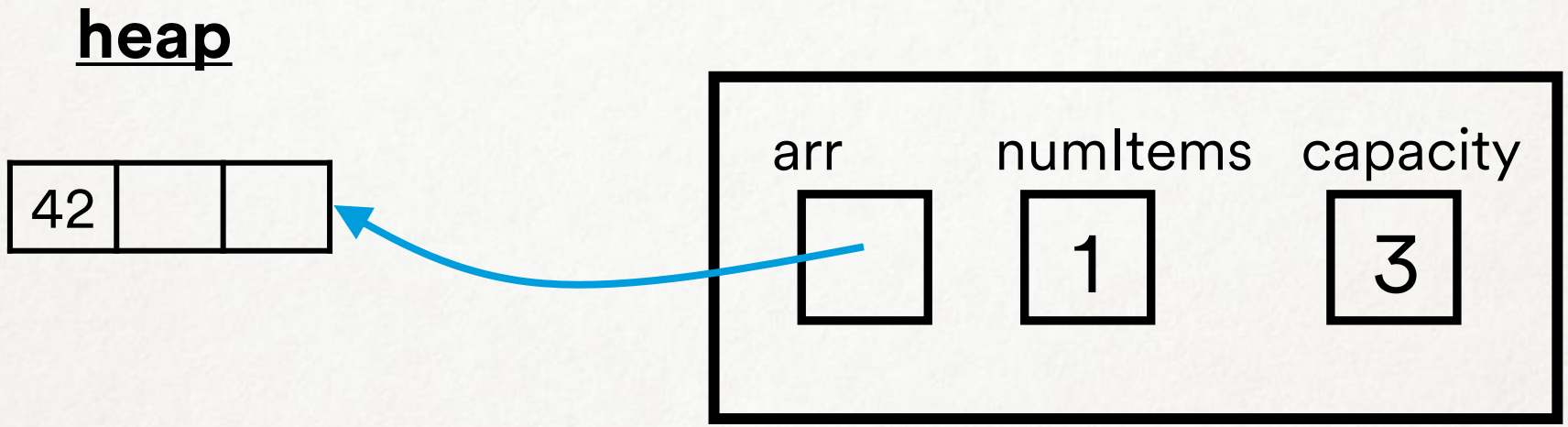
int& at(int index);

- The Vector's .at(index) function should return the value held at that index in `arr`
  - But, it should be a reference so it can be changed! You don't have to do anything extra—just return the value at the index. It'll be given back as a reference thanks to the return type!

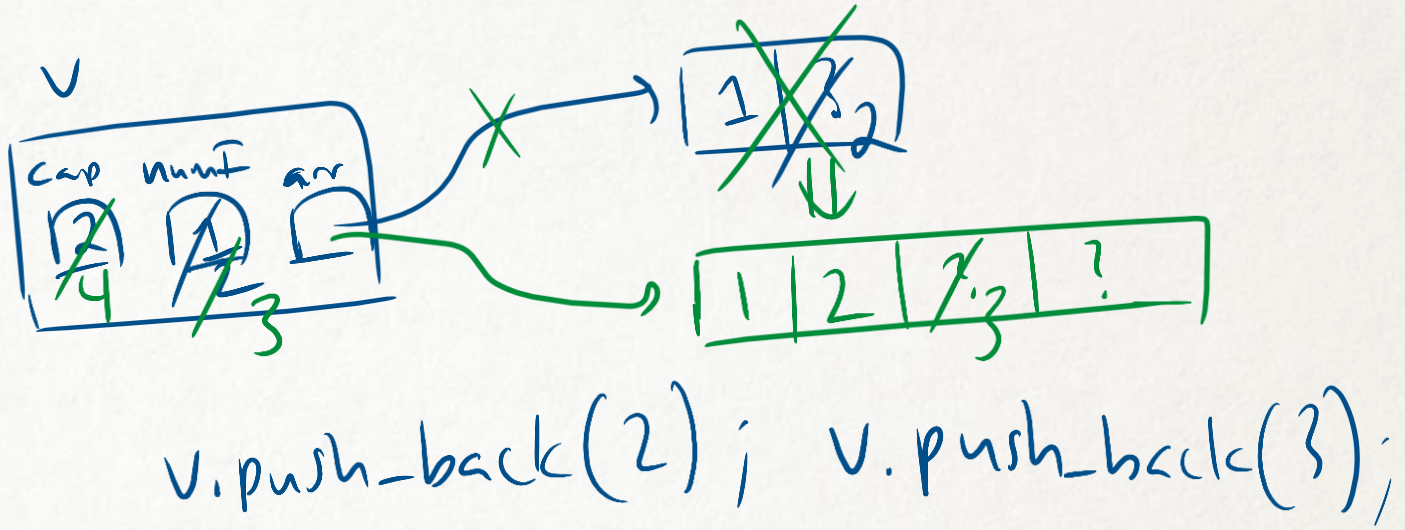


.size(...)

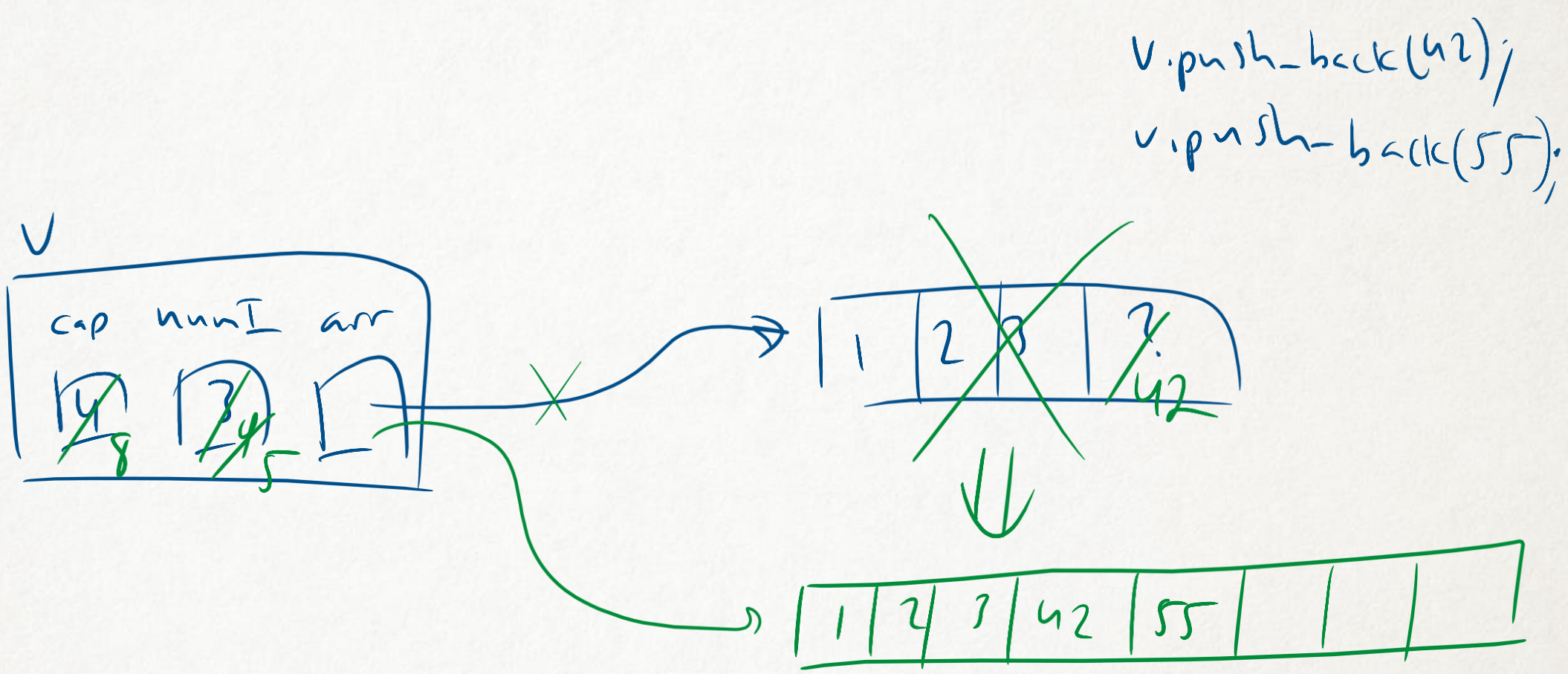
- This is just a getter: return our `numItems`
- We're kind of tricking the user to think that there's just an array with 1 thing (in the example below), when in reality it has room for two more things.



.push_back()



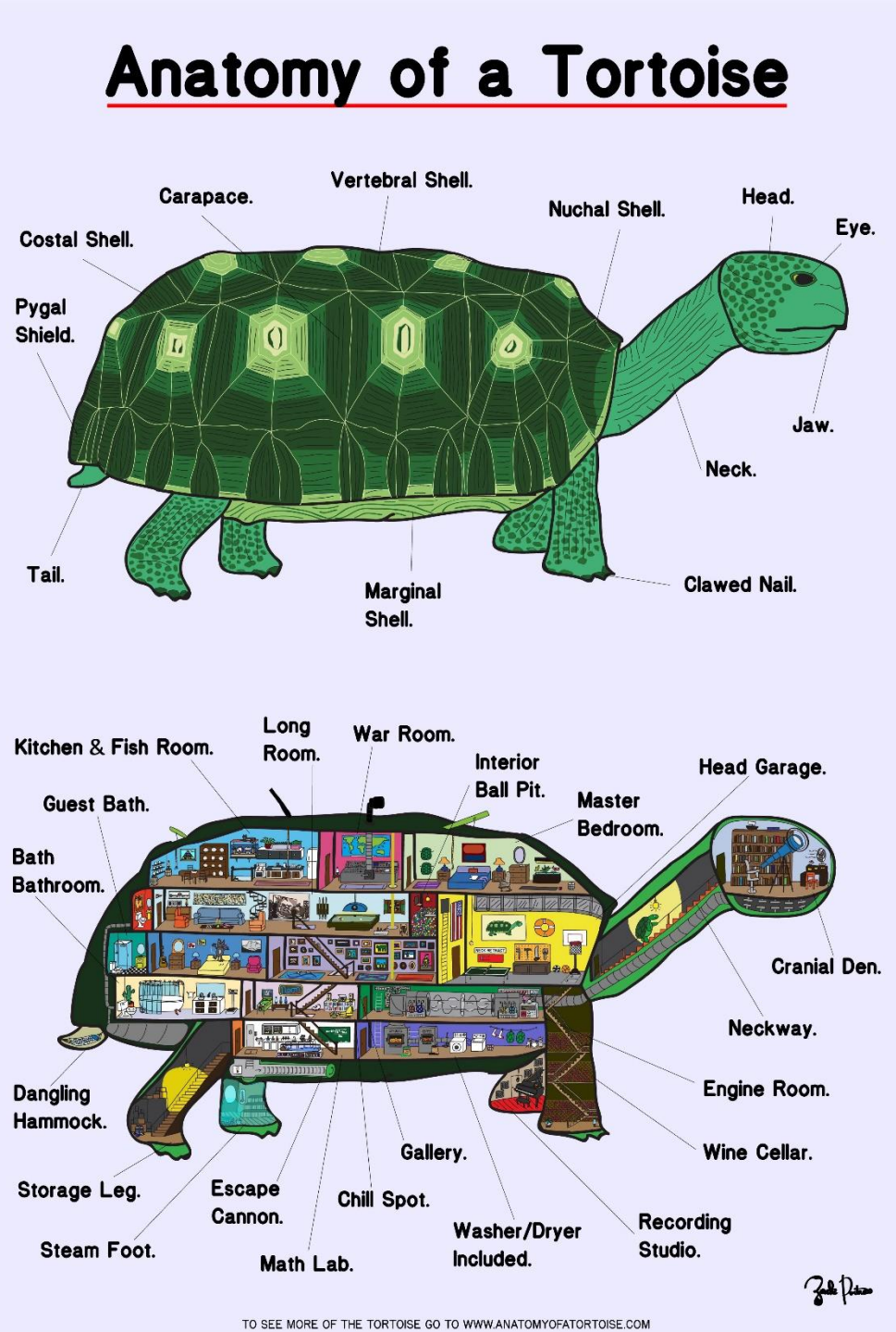
- This one is the hard one
- We might run out of space!
- **Case 1**: If we can safely add one more thing to our current array, then do it
  - Increment numItems, too, of course
- **Case 2**: If we don't have the capacity to add anything else, here's the secret sauce:
  - Create a new array on the heap with ***double*** our current capacity
  - Copy all the existing elements into that
  - Reset `capacity` appropriately
  - Add the next element using the rules for Case 1


v.push_back(42);
v.push_back(55);
v
cap
numI
arr
4
8
3
4
5
1
2
3
3
42
1
2
3
42
55

To be continued...

ADT from the user's perspective:

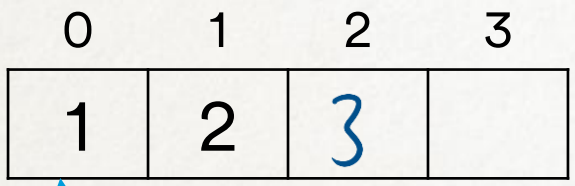
ADT from the library implementer's perspective:



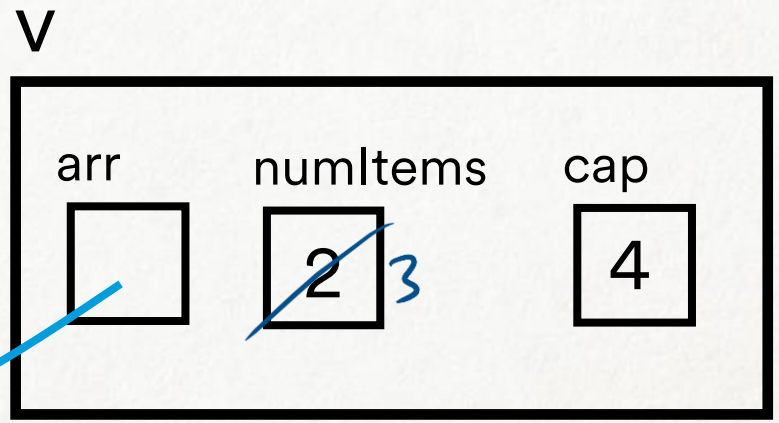
Crushed ICE

- push_back Case 1 example
  - Given this state of memory, tell me what changes

Heap:



Stack:

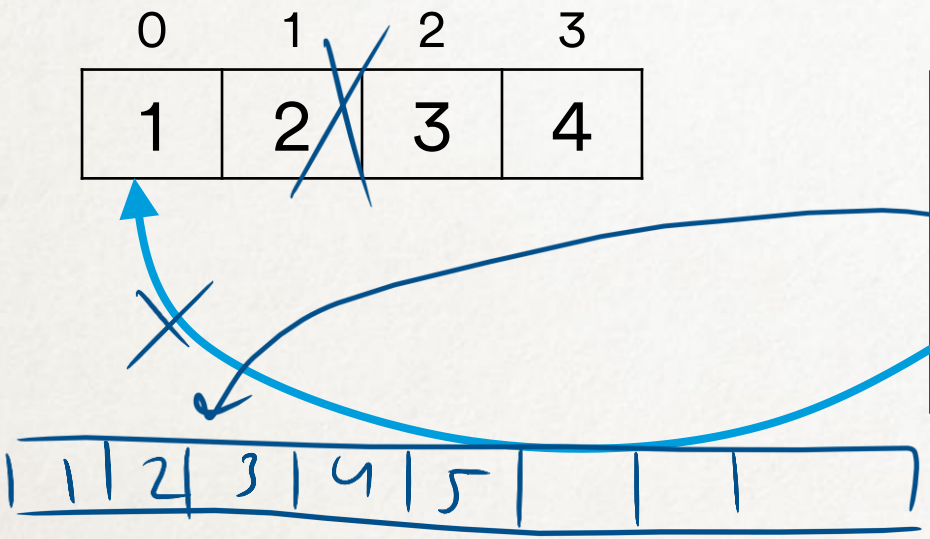


v.push_back(3)

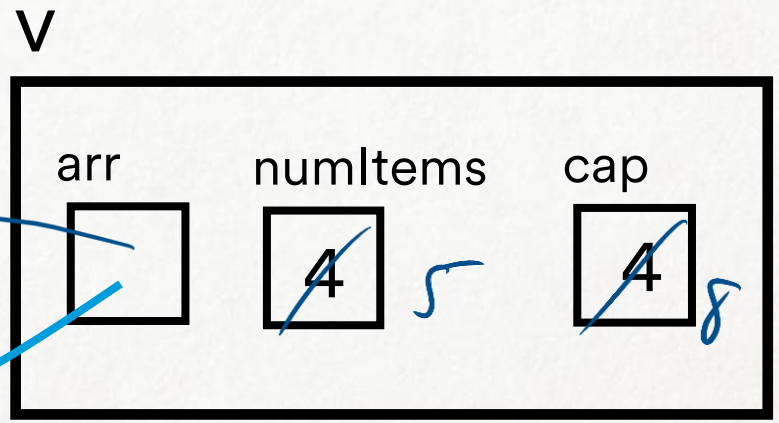
Crushed ICE

- push_back Case 2 example
  - Given this state of memory, tell me what changes

Heap:



Stack:



v.push_back(5)

Running time complexity: A taste of what's to come

How fast is it to do each of the following? Which are faster than others?

.size() — just a var lookup. Super fast! One line of code. **Constant time.**

.at() — access arr at a certain index. One line of code. Two "steps".
Constant time.

.push_back()

Case 1: ~3 lines of code. **Constant time.**

Case 2: depends on the size of old array. Large array = lots of time.
~N "steps". **Linear time.**

We'll make this precise very soon

On average, .push_back() is still constant time

Surprising, huh?

Space complexity: A taste of what's to come

How much memory does it take to do each of the following?

.size() — no args to store, returns a single int. 1 int = 4 bytes. **Constant space.**

.at() — 1 int parameter, returns 1 int. ~2 ints of space. **Constant space.**

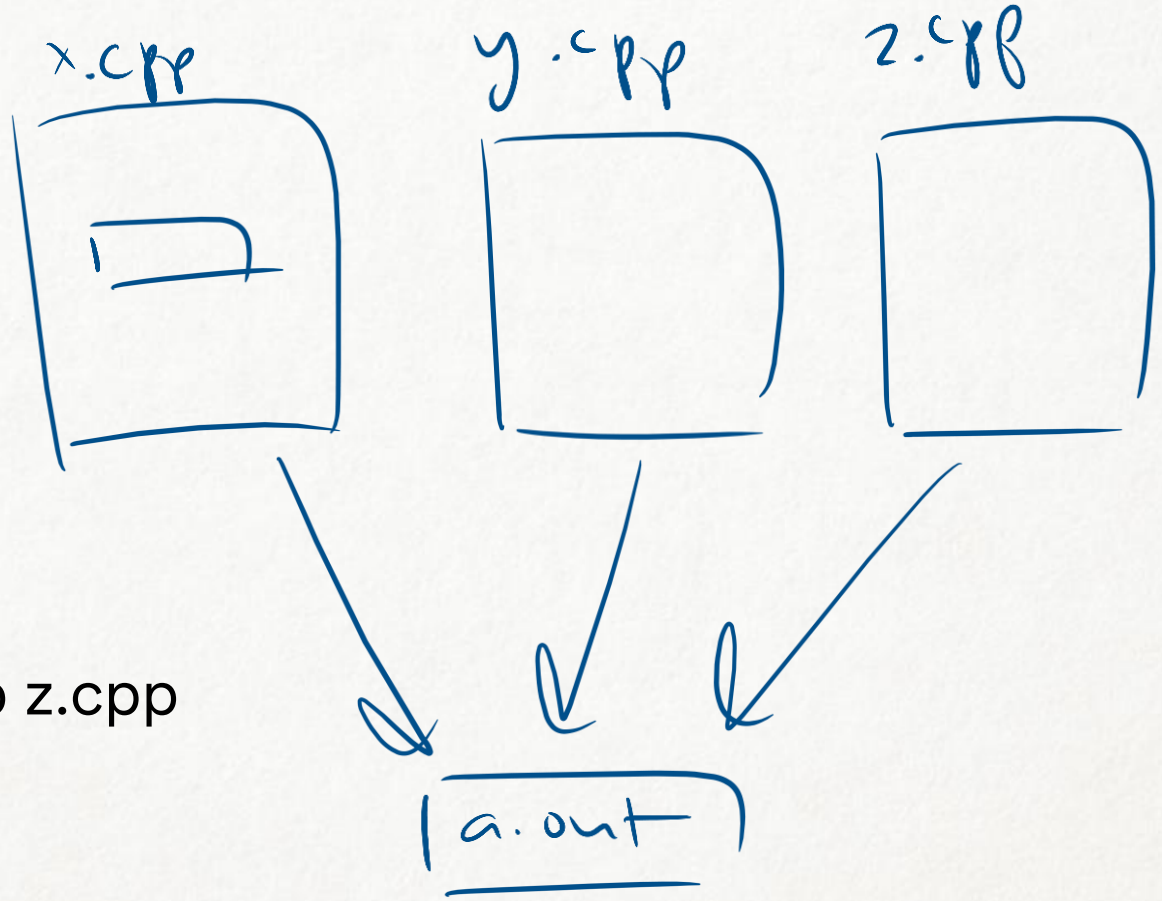
.push_back()

Case 2: allocate new array of size 2N, a few int params/vars.

2N + C. **Linear space.**

Separate compilation & Makefiles

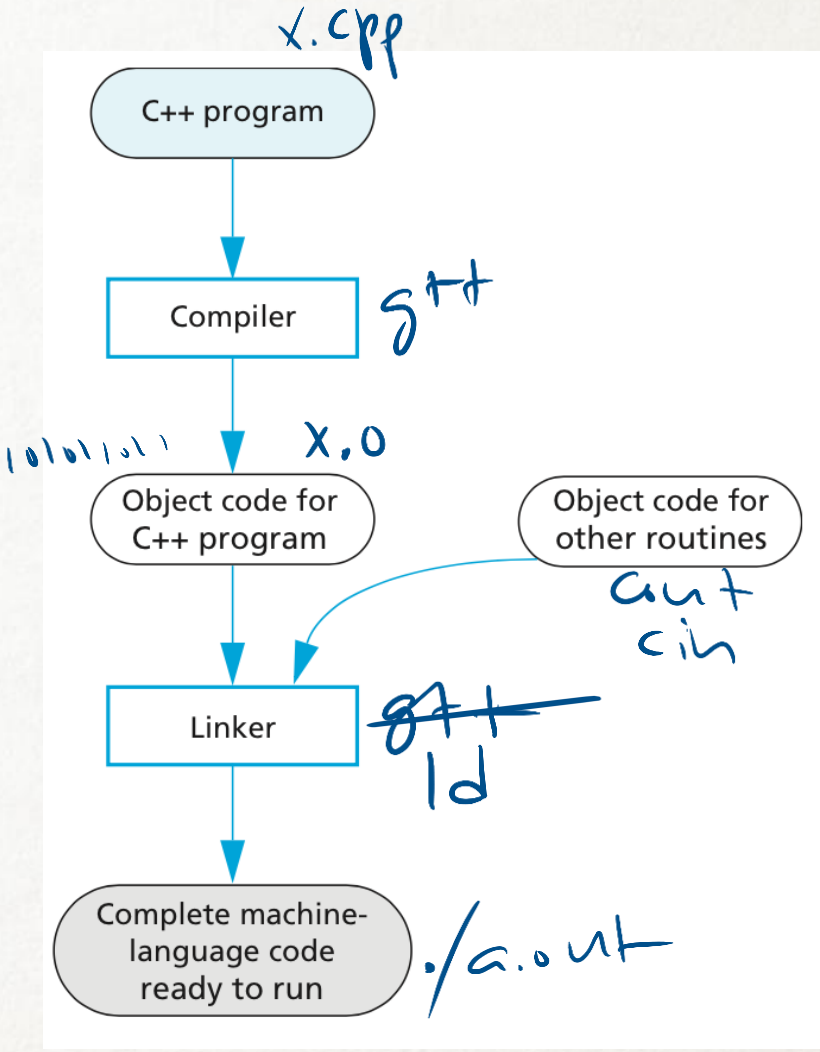
If I change just one file, I have to recompile them all?



g++ x.cpp y.cpp z.cpp

Review: Compiling and Linking

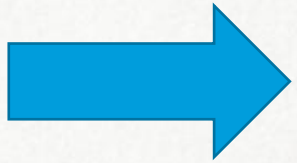
- A **compiler** translates a single program to **machine code / object code**
- A **linker** combines programs together so they can share functionality
  - **Libraries**: pre-implemented code that makes your life easier
- After compilation and linking, you have a machine code program that you can run



No! That's inefficient!

(Future Lawton, use the Counter class as an example)

Use object files as an intermediate stage

g++ -c x.cpp  x.o

↑ *Just one file!*

You only have to recompile what you changed

x.o is the compiled (machine code) form of just x.cpp

To link, you say:

```
g++ x.o y.o … -o exec_file_name
```

Compilation was the bottleneck. Linking is fast!

Makefiles automate the process of (re)compiling

A Makefile is a file called “Makefile” that contains *recipes* for how to compile your programs

Example Makefile:

```
# recipes for making main.o and foo.o
main.o: main.cpp
	g++ -c main.cpp

foo.o: foo.cpp
	g++ -c foo.cpp

# link main.o and foo.o together in prog
prog: main.o foo.o
	g++ main.o foo.o -o prog
```

Makefile'ing a PB&J sandwich

pb-j : plate knife PB J bread
_TAB --> open PB jar
open J jar
take out 2 slices of bread
...
close sandwich

PB: car money
--> go to store
buy PB

Makefile structure

requirements must exist before the body of the rule can run

```
thing_you_want: requirement_1 requirement_2 ...
    cmd1 to partially make thing
    cmd2 to partially make thing
    ...
    g++ file1 file2 –o thing_you_want
```

target

steps needed to make the thing

To "run" a Makefile, you say:

```
make targetname
```

or

```
make
```

↑ *this will run the first rule*

Shorthand and variables

$@ — the target
$^ — all the ingredients/requirements
$< — the first ingredient
X=def — variable definition (space around = doesn’t work)
$(X) — variable access

We like to add a “clean” rule to remove all compiled files

Comparison Classes

Let's make a class and std::sort it

- A Holder class that holds a single int
- C++ complains because there's no way to compare our class
- We will give it a way!
- See std::sort docs for more details
- Option 1: make an operator<
- Option 2: make a comparison function
  - Takes two Holders and returns true when the first should come before the second in sorted order
- Option 3: make a **comparison class** (a.k.a. **functor**)
  - A class that has a bool operator() implementation that takes two Holders and returns true when the first should come before the second in sorted order